Query Rewrite Optimization for NebulaStream

Daniel Dronov
Riccardo Marin
Tobias Engelbrecht

TU Berlin

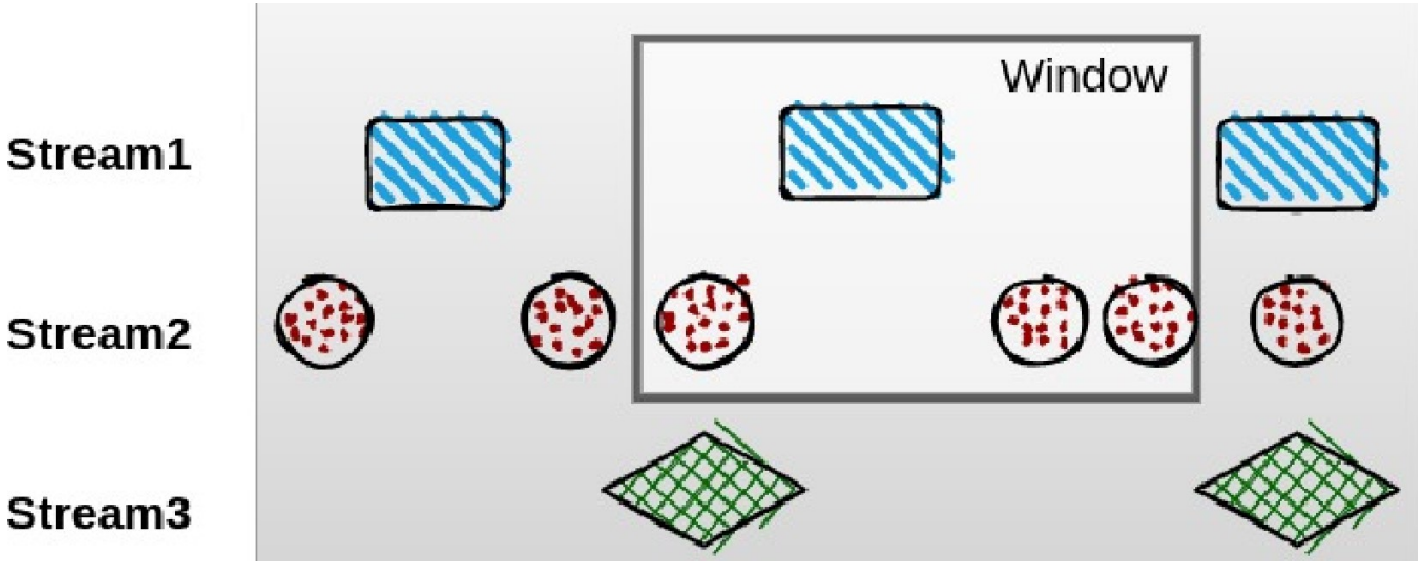
17 July 2023

Agenda

1. Background
   - Stream processing systems
   - Query plans
   - Project goal
2. Solution approach
   - Re-write rules
3. Benchmarking
   - FilterPushDownRule overhead
   - Benchmarking queries
   - Results evaluation
4. Future work
5. Project reflections

Stream processing systems

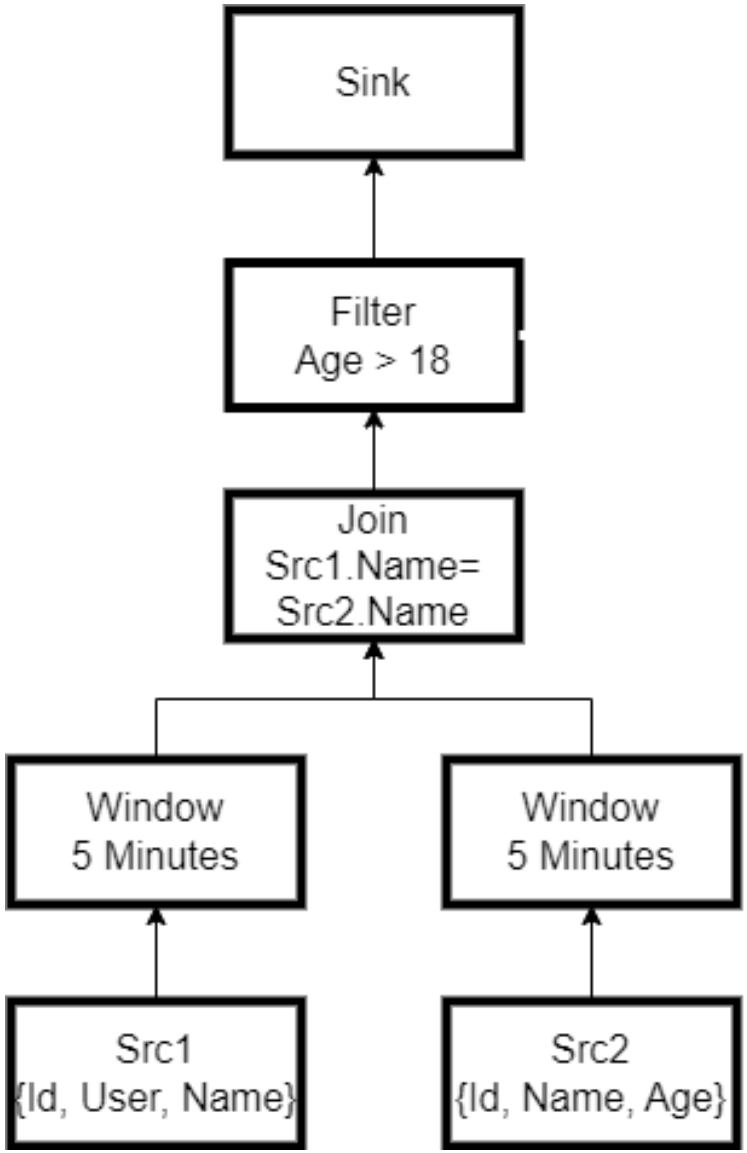
- Different sources produce data streams with well-defined schemas
- Windows are used to discretize data from the stream
- SQL-like queries are used to process the data
  - filter, map, projection, join, union, aggregations, **window**



- *NebulaStream* is a stream processing system designed the IoT

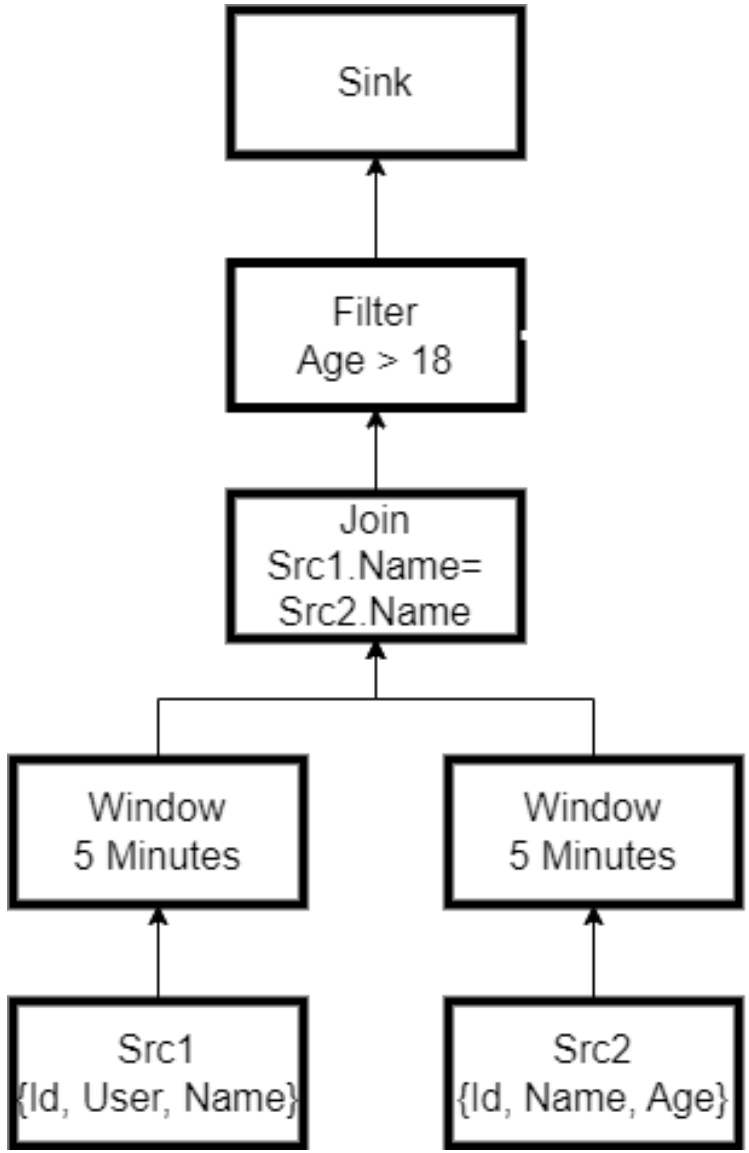
Query plans

- Logical Query Plans represent queries.



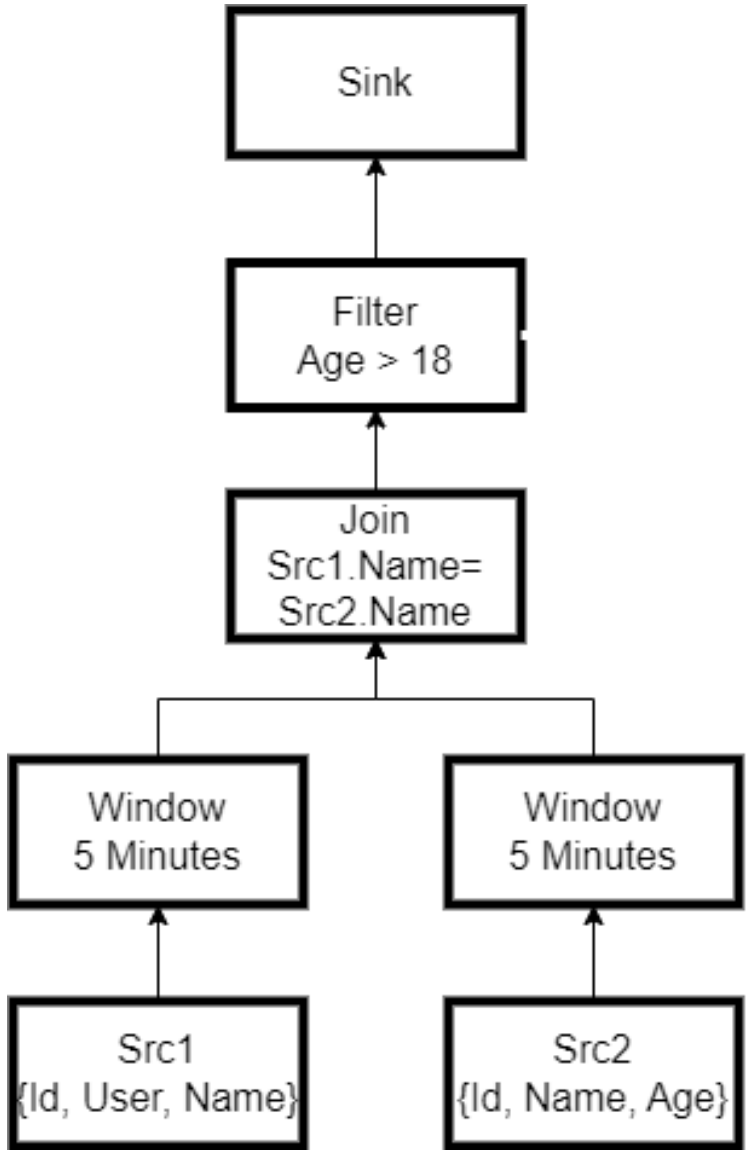
Query plans

- Logical Query Plans represent queries.
- Optimizer rewrites the query



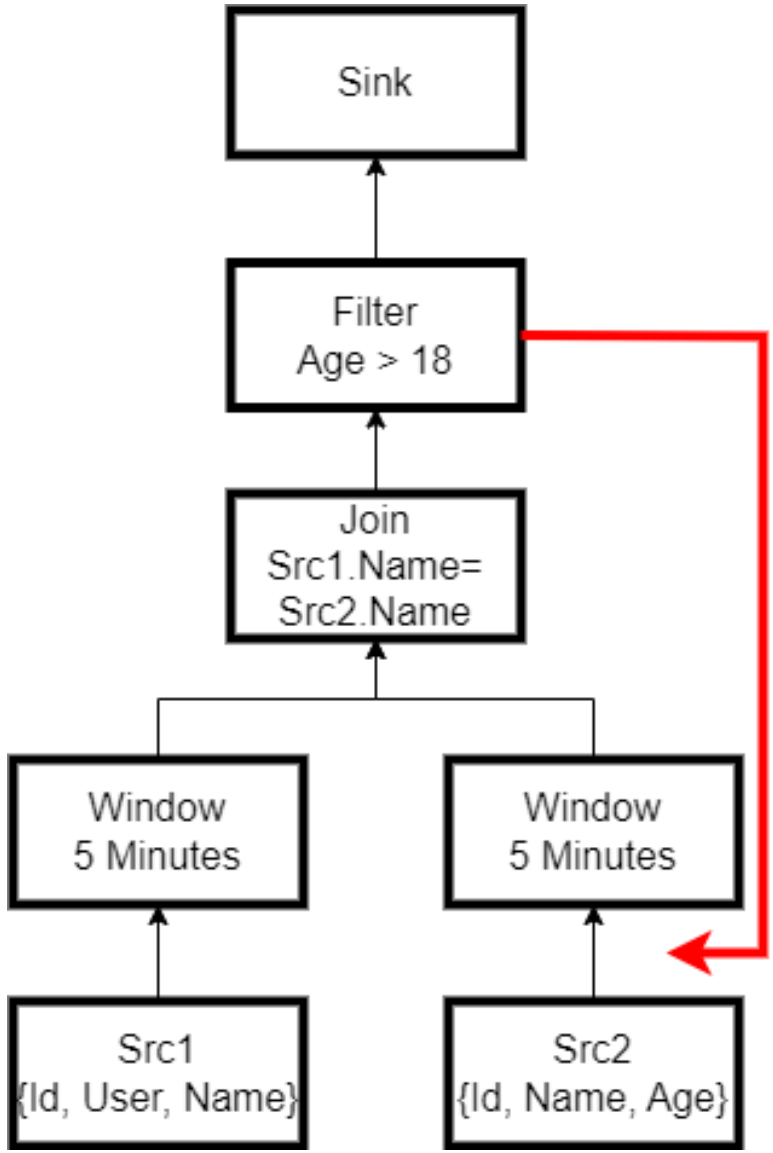
Query plans

- Logical Query Plans represent queries.
- Optimizer rewrites the query
- → Reduce intermediate results for each processing step



Query plans

- Logical Query Plans represent queries.
- Optimizer rewrites the query
- → reduce intermediate results for each processing step



Project goal

Project goal

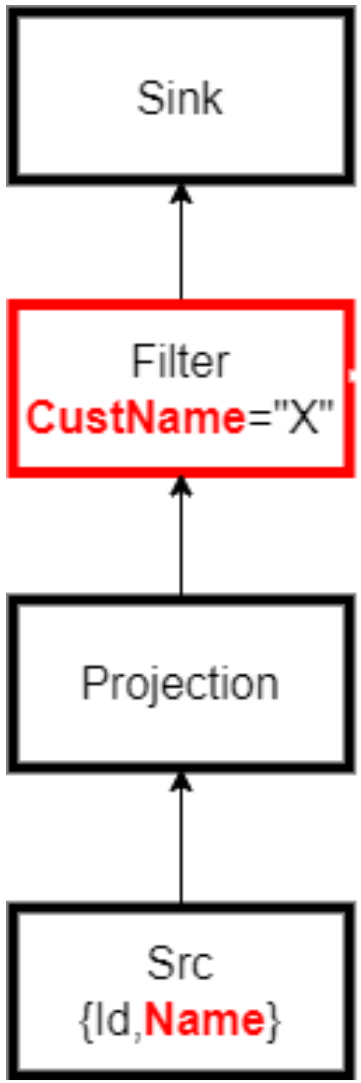
How can query re-write rules be applied to the query plans?
What benefits and limits do they entail?

Solution approach

- Analyze rewrite rules and add them to the query optimizer
  - FilterPushDownRule
  - PredicateReordering
  - FilterSplitUp
- Verify the correctness of our implementation with unit tests and provided integration tests
- Measure the performance benefits with benchmarks

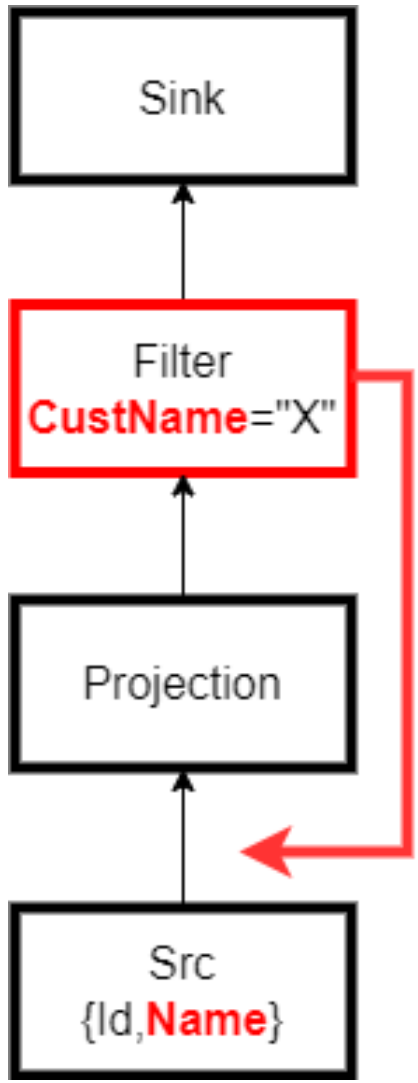
Filter push-down below projection

- Boost performance, as other operators have to iterate over less tuples
- The amount of tuples is more important than the memory size of the data
- Projections can rename columns in NebulaStream



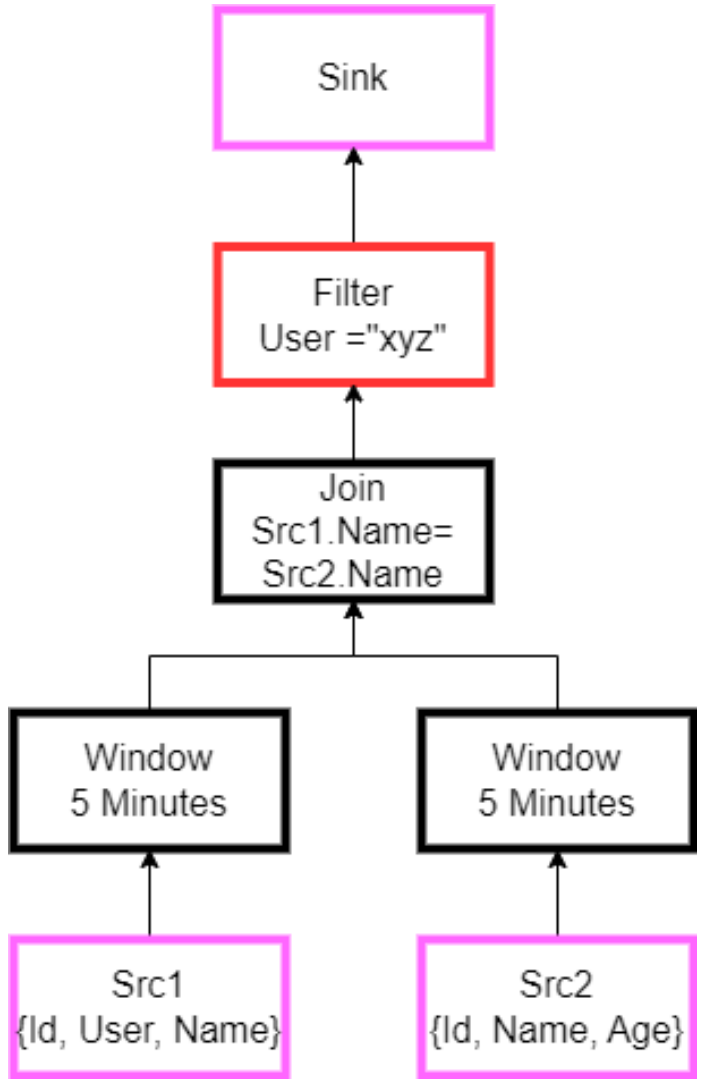
Filter push-down below projection

- Can boost performance, as other operators have to iterate over less tuples
- The amount of tuples is more important than the memory size of the data
- Projections can rename columns in NebulaStream



Filter push-down below join

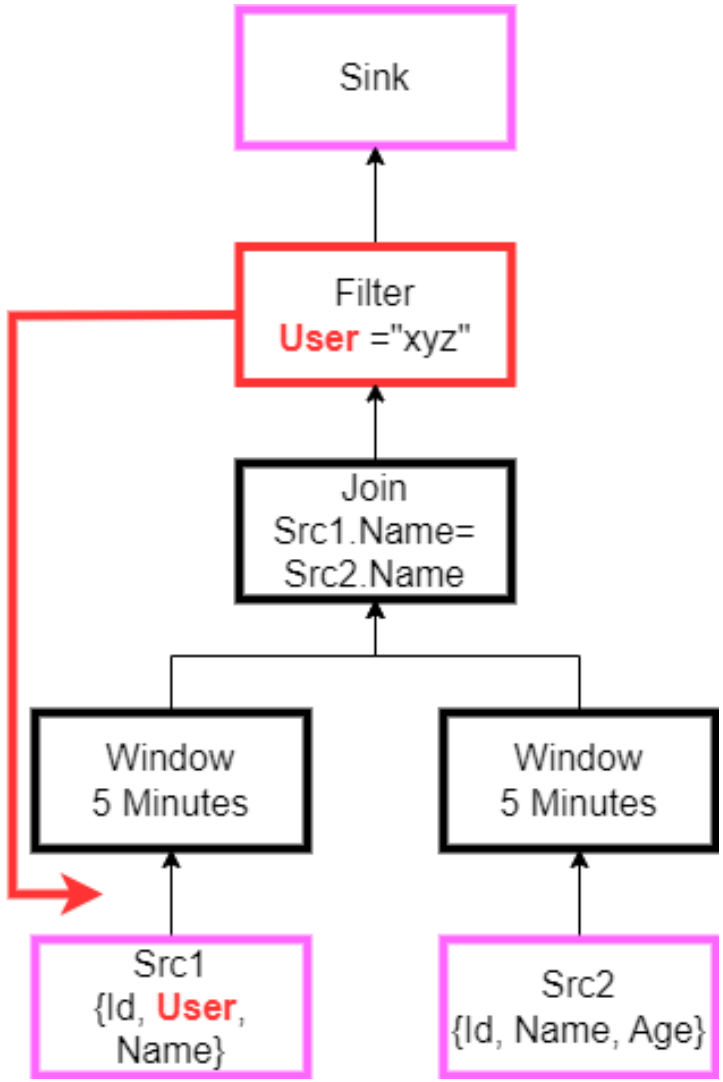
- Pushing a filter below a join reduces the intermediate results to join on
- Joins need windows in the context of streams
- Filters work on tuples, so they can be pushed below the windows
- There are three base cases



Filter and Join example

Filter push-down below join

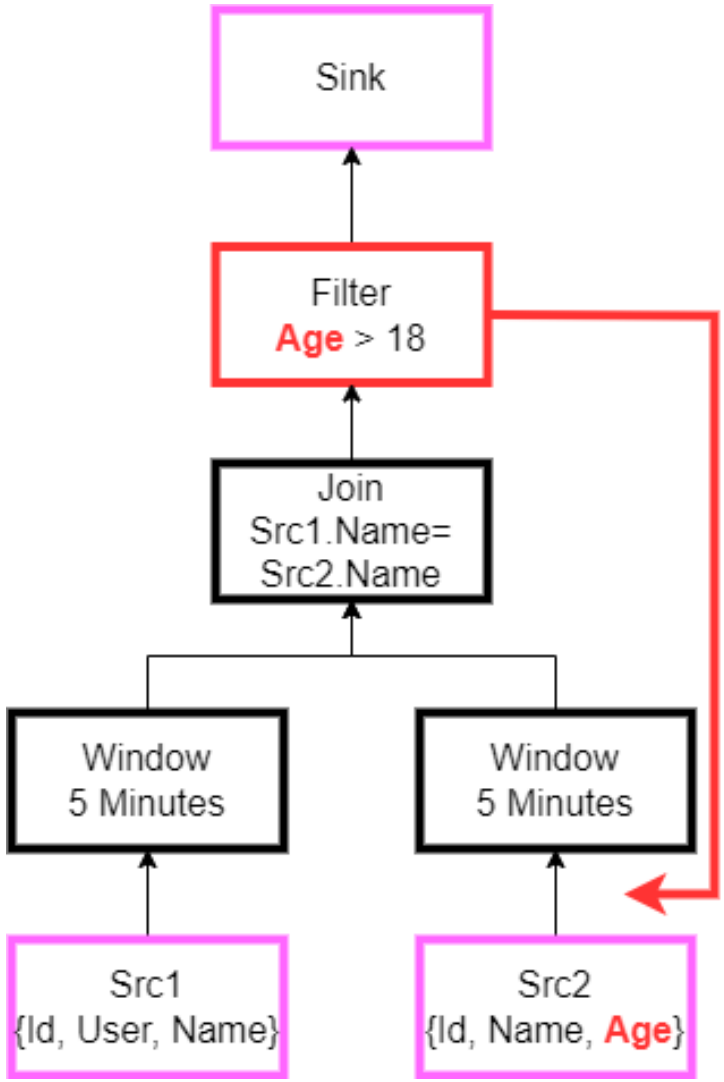
- Pushing a filter below a join reduces the intermediate results to join on
- Joins need windows in the context of streams
- Filters work on tuples, so they can be pushed below the windows
- There are three base cases
  - First case: push-down to left branch



Filter and Join example

Filter push-down below join

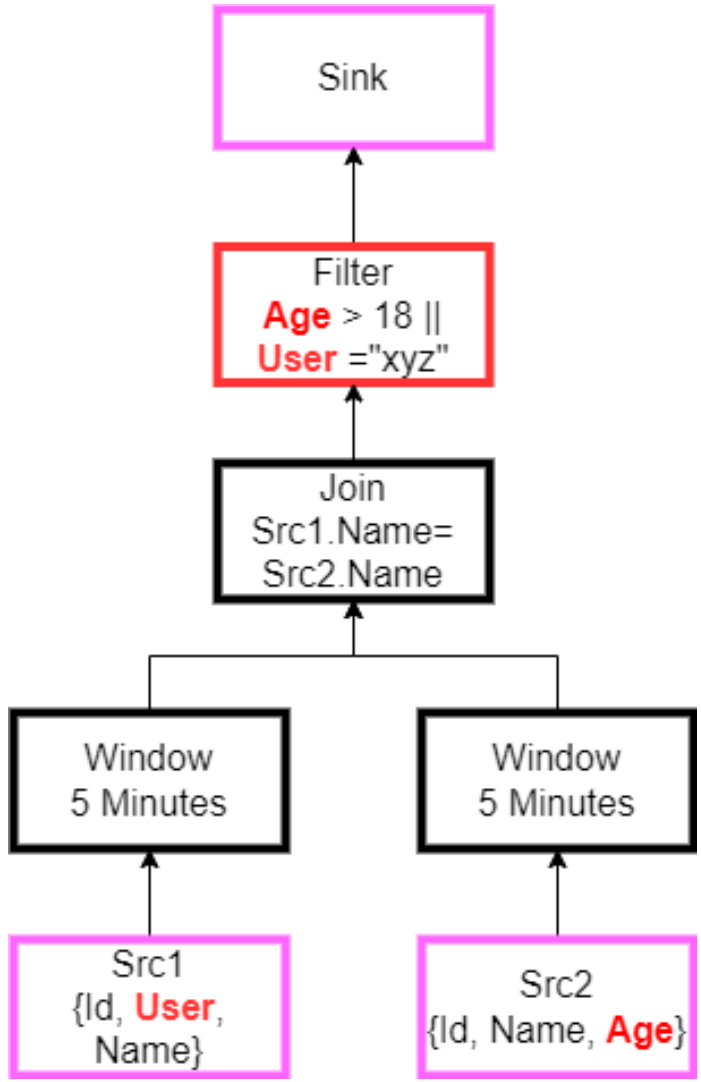
- Pushing a filter below a join reduces the intermediate results to join on
- Joins need windows in the context of streams
- Filters work on tuples, so they can be pushed below the windows
- There are three base cases
  - Second case: push-down to right side



Filter and Join example

Filter push-down below join

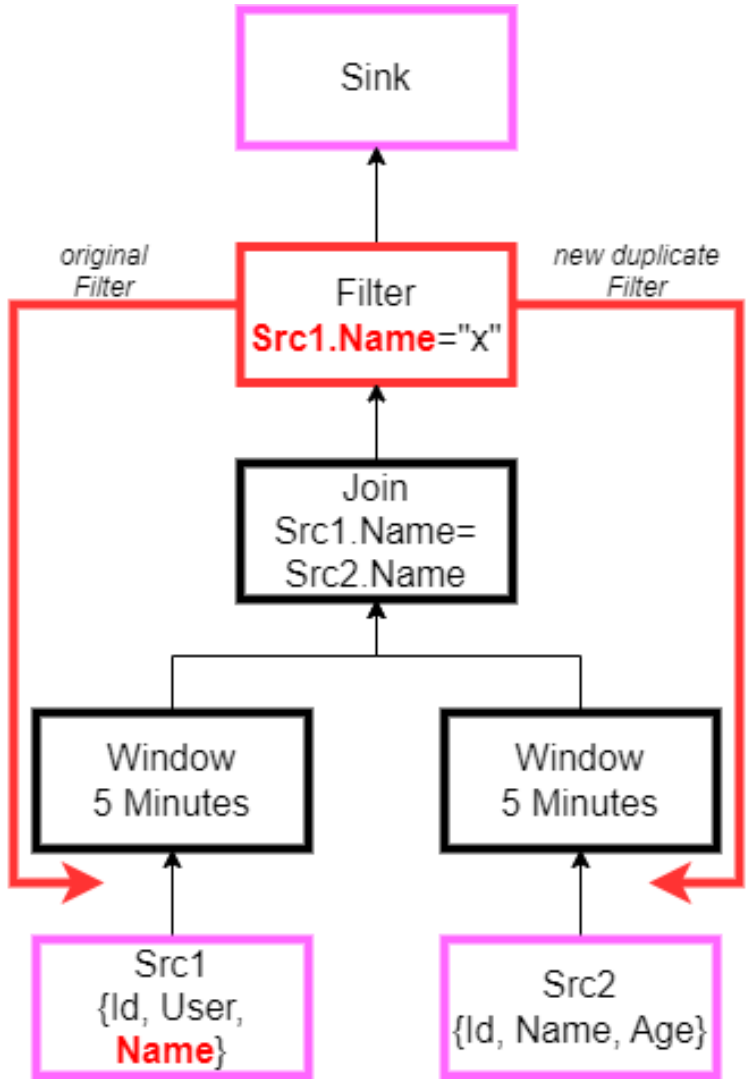
- Pushing a filter below a join reduces the intermediate results to join on
- Joins need windows in the context of streams
- Filters work on tuples, so they can be pushed below the windows
- There are three base cases
  - Third case: can't push-down



Filter and Join example

Filter push-down below join

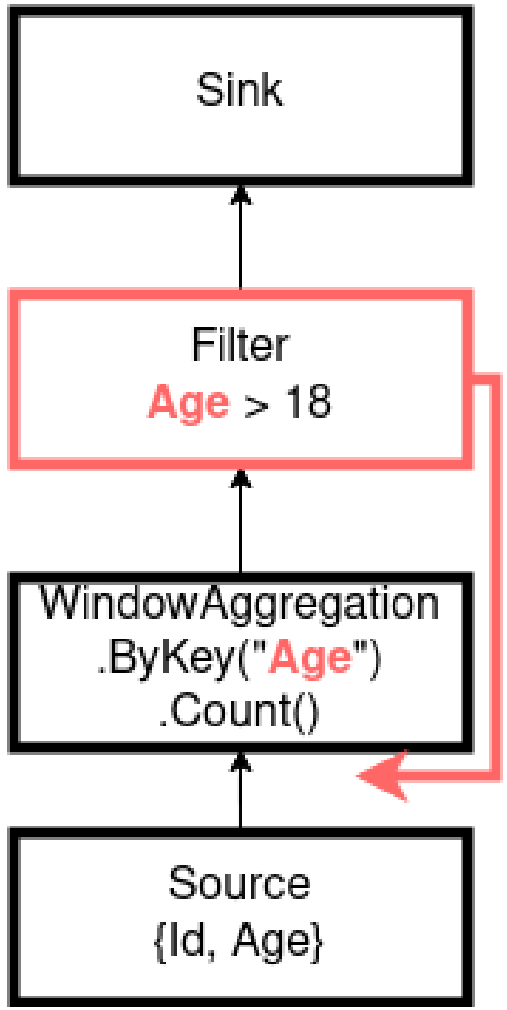
- Pushing a filter below a join reduces the intermediate results to join on
- Joins need windows in the context of streams
- Filters work on tuples, so they can be pushed below the windows
- There are three base cases
  - special case: predicate part of join-condition



Filter and Join example

Filter push-down below window aggregation

- In Nebulastream, usages for windows: aggregations and joins
- For joins, the push-down below join rule applies
- For aggregations, we can push-down if it does not impact the aggregation result
  - If there is a group by clause and filter on the same attribute, it can be pushed down



Push-down constraints

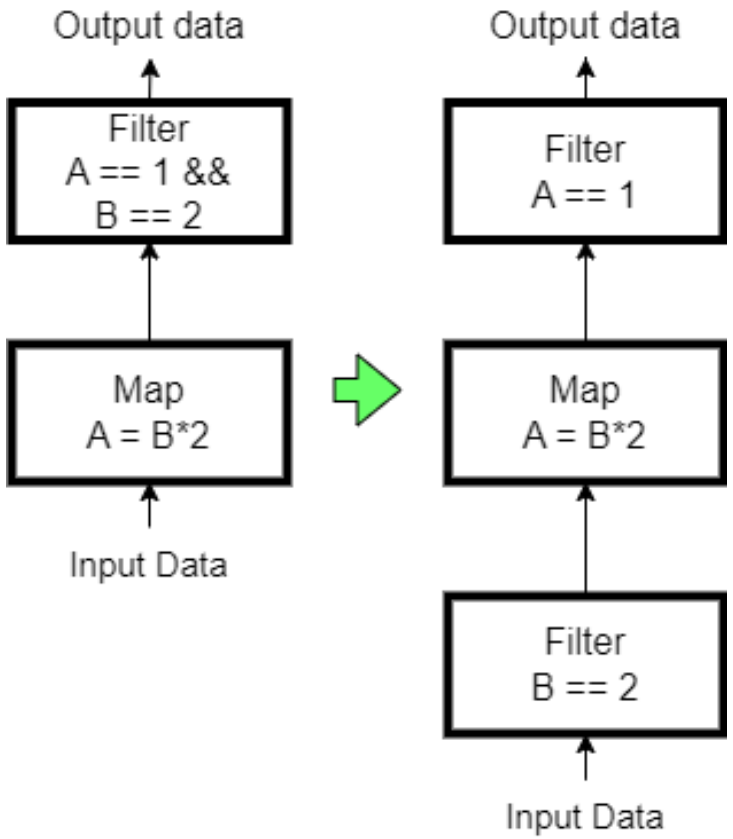
- Pushing down through union and map also possible, but have their own constraints
- Pushing down filters has a computational cost to take into account
- Might not be worth when the filter selectivity is low
  (= the filter is not greatly reducing the number of returned tuples)
- We also need to consider the operator fusion done in the compiling phase

The measured push-down overhead is low and therefore we always push down the filters

Filter predicate split up

This rule handles filter predicates with conjunctions (”and”).

- The conjunctions are converted into consecutive FilterOperators
- This can potentially allow pushing one predicate below the join
- De Morgan’s laws to reformulate negated disjunctions



Filter reordering

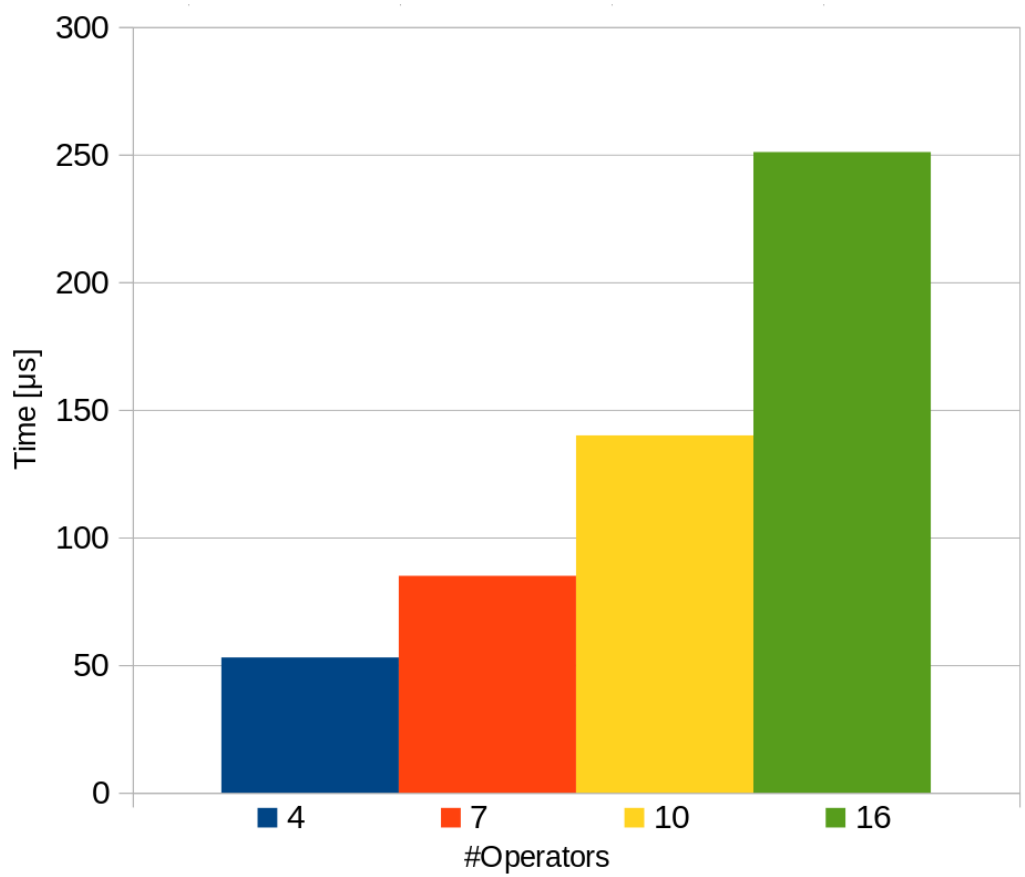
- Identify consecutive filters
- The filters are sorted by selectivity
- → the predicates with an high selectivity are executed first
- This way the query plan received by the compiler is already optimal:

```
if (p1 && p2) // Selectivity p1 >> p2
```

FilterPushDownRule overhead

We computed the overhead of the execution of the FilterPushDownRule

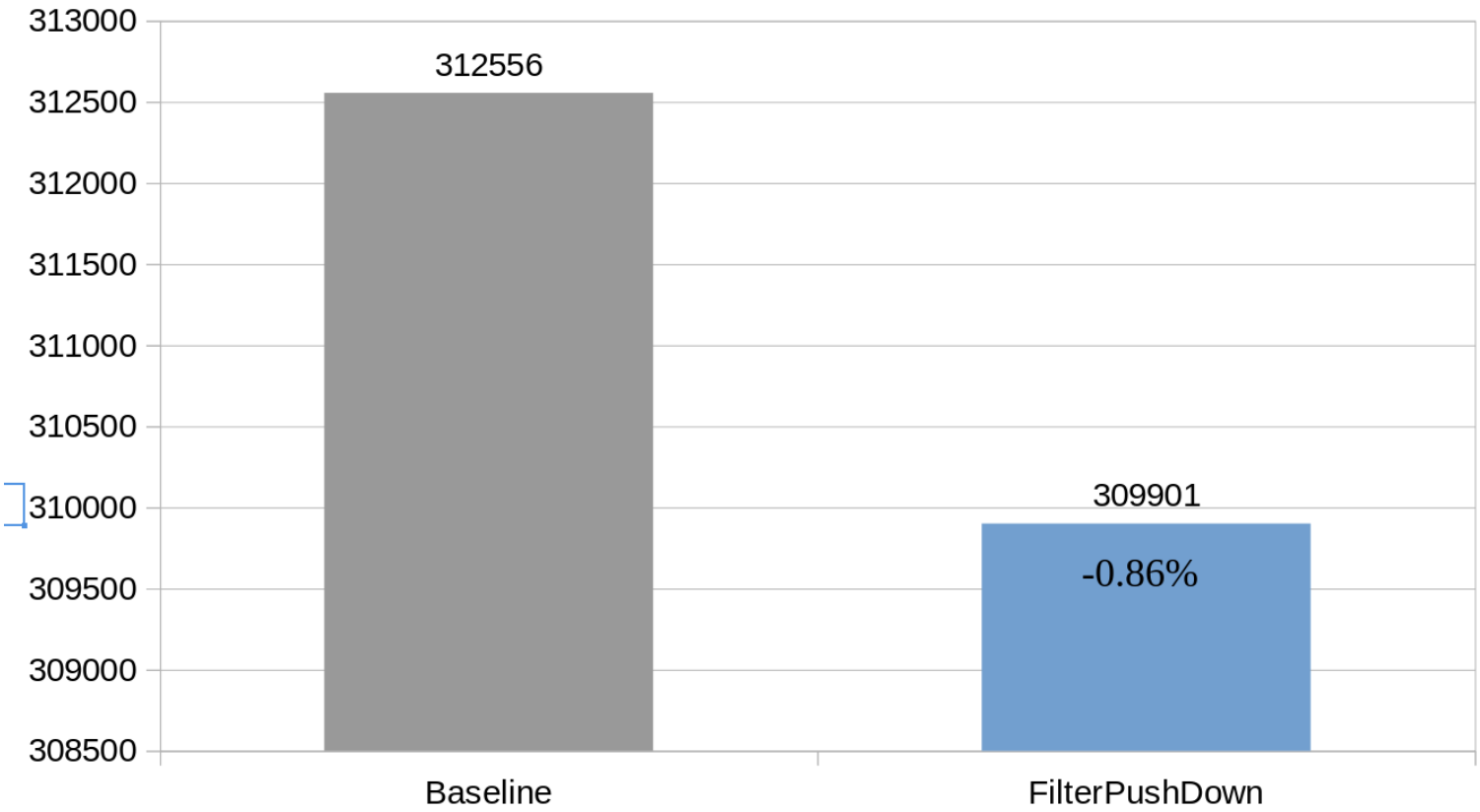
- Measured the average time to apply the rule on the unit tests
- The overhead is relatively small (in the order hundreds of microseconds)



Benchmarking queries

Query 1

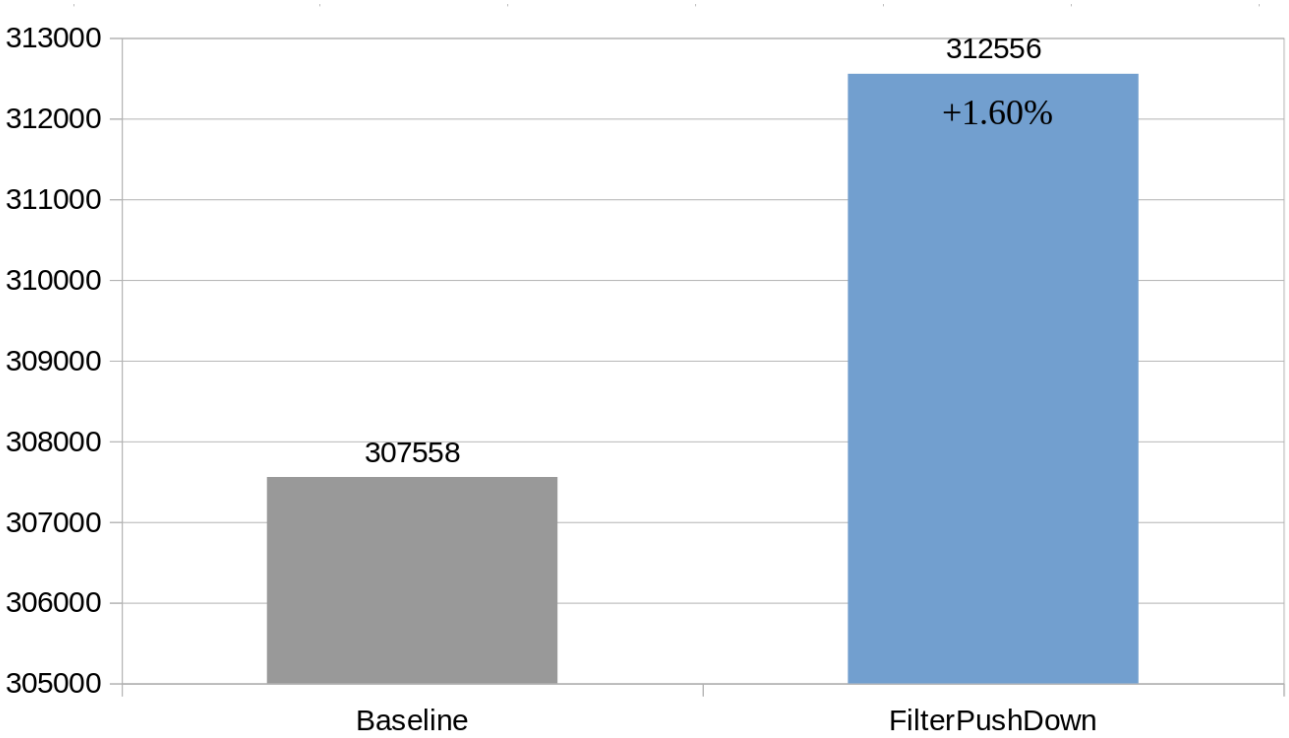
```
Query::from("input1")
    .filter(Attribute("id") < 45 && Attribute("id") > 2)
    .filter(Attribute("timestamp") == 100)
    .filter(Attribute("timestamp") == 2 && Attribute("id") > 2)
    .sink(NullOutputSinkDescriptor::create());
```



Benchmarking queries

Query 2

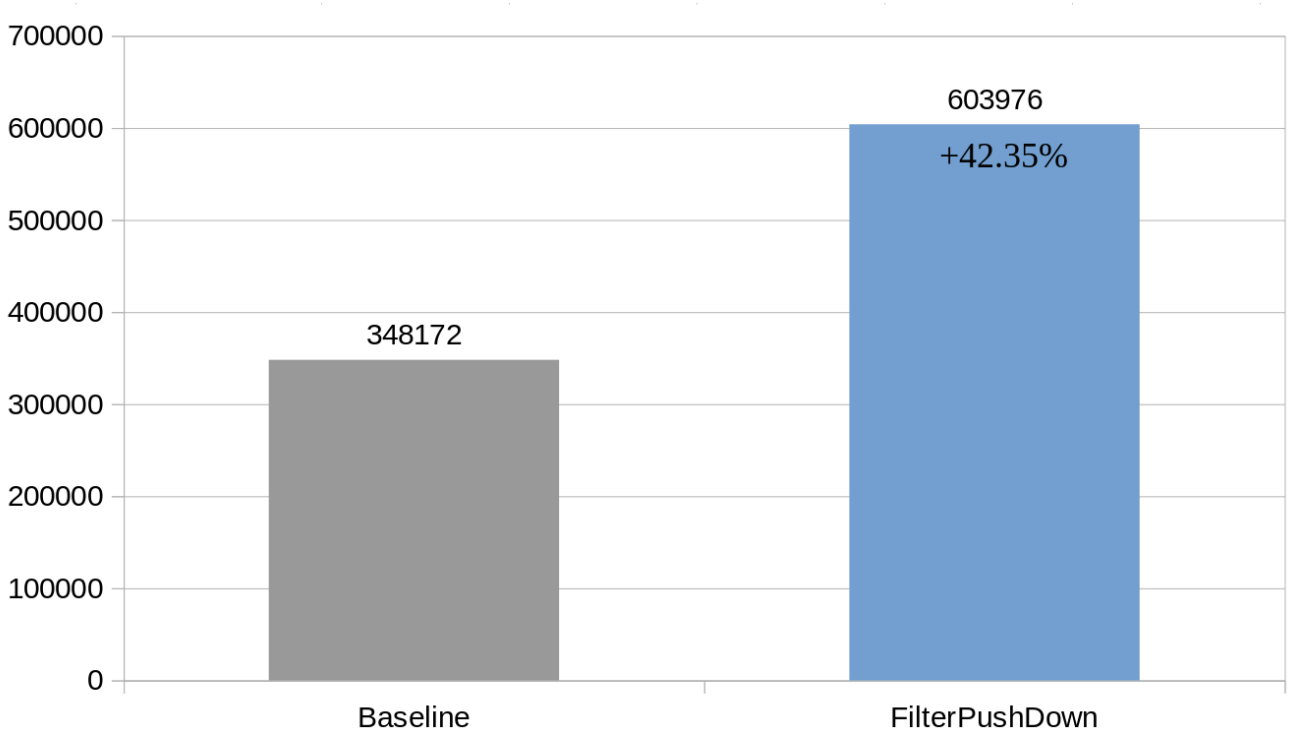
```
Query::from("input1")
    .filter(Attribute("id") > 25)
    .map(Attribute("timestamp") = Attribute("timestamp") - 10)
    .filter(Attribute("id") > 45)
    .map(Attribute("id") = Attribute("id") * 2)
    .filter(Attribute("timestamp") < 55)
    .sink(NullOutputSinkDescriptor::create());
```



Benchmarking queries

Query 3

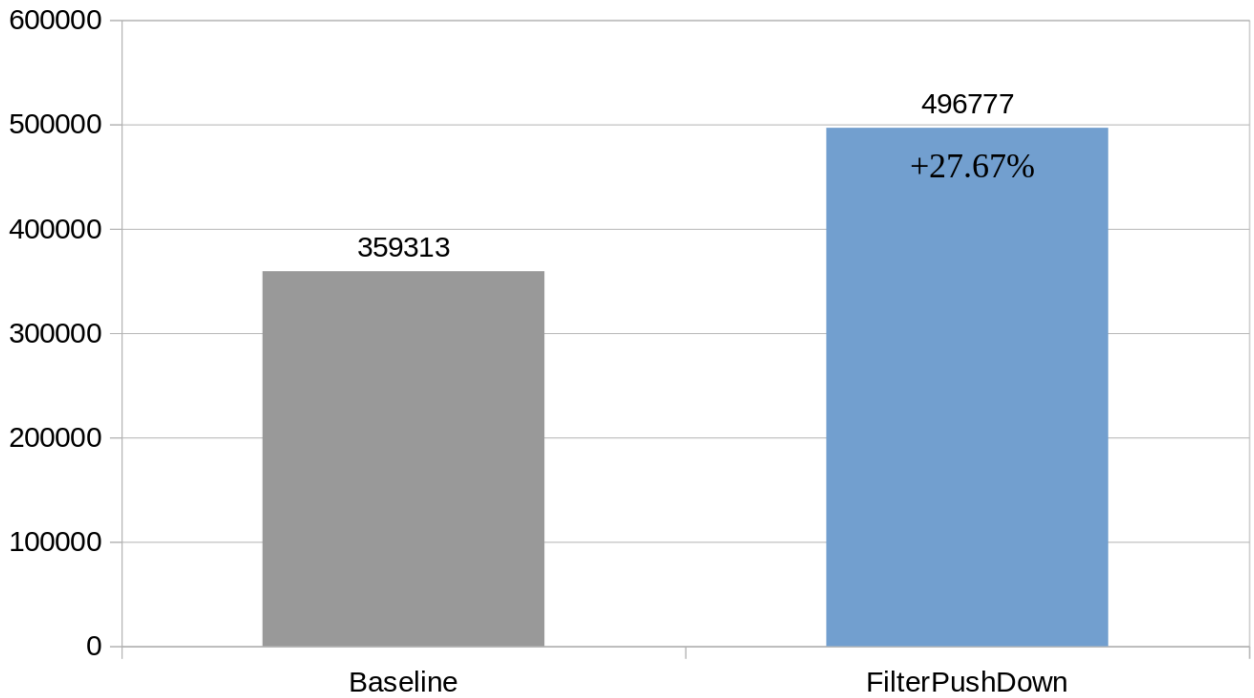
```
Query::from("input1")
    .joinWith(Query::from("input2"))
    .where(Attribute("value"))
    .equalsTo(Attribute("value"))
    .window(TumblingWindow::of(EventTime(Attribute("timestamp")), Milliseconds(500)))
    .  filter  (Attribute("value")  < 45)
    .sink(NullOutputSinkDescriptor :: create());
```



Benchmarking queries

Query 4

```
Query::from("input1").joinWith(Query::from("input2"))
. filter (Attribute("value") > 5))
.where(Attribute("value")).equalsTo(Attribute("value"))
.window(TumblingWindow::of(EventTime(Attribute("timestamp")), Milliseconds(500)))
. filter (Attribute("value") > 20)
. project (Attribute("id"), Attribute("value"))
. filter (Attribute("value") > 10)
.sink(NullOutputSinkDescriptor :: create());
```



Results evaluation

- Benchmarks showed relatively high variance among the repeated execution runs
- FilterPushDownRule can be used to effectively optimize the queries, increasing the throughput by a significant amount
- PredicateReordering needs a selectivity value to work properly

Future work

Duplicate operator elimination rule

- We assume it can be impactful on the execution time
- Identification of redundant operators in the query plan
- Can be found by comparing operators of the same type
- Elimination of the unnecessary operators

Project reflections

What went well

- Managed to speed up the queries by adding new re-write rules
- Units tests enabled us to quickly debug and test our code
- Good and productive collaboration within the team

Challenges

- Dealing with production level C++ code
- Size and complexity of the codebase
- Review process slowed down by the CI pipeline execution time

Sources

- Slide 3 figure: https://docs.nebula.stream/docs/query-api/complexeventprocessing/